Node configuration for the Aho-Corasick algorithm in Intrusion Detection Systems

Alexsandre B. Lacroix, J.M. Pierre Langlois, François-Raymond Boyer,
Antoine Gosselin and Guy Bois
Computer and Software Engineering Department
Polytechnique Montréal, Canada
{alexsandre.lacroix, pierre.langlois, francois-r.boyer, antoine.gosselin, guy.bois}@polymtl.ca

ABSTRACT
In this paper, we analyze the performance and cost trade-off from selecting two representations of nodes when implementing the Aho-Corasick algorithm. This algorithm can be used for pattern matching in network-based intrusion detection systems such as Snort. Our analysis uses the Snort 2.9.7 rules set, which contains almost 26k patterns. Our methodology consists of code profiling and analysis, followed by the selection of a parameter to maximize a metric that combines cycles count and memory usage. The parameter determines which of two types of nodes is selected for each trie node. We show that it is possible to select the parameter to optimize the metric, which results in an improvement by up to 12× compared with the single node-type case.

Categories and Subject Descriptors
I.5.4 [**Pattern Recognition**]: Applications – *Text processing*.

C.2.0 [**Computer-Communication Networks**]: General – *Security and protection*.

General Terms
Algorithms, Performance, Design, Experimentation, Security.

Keywords
Aho-Corasick algorithm, node configuration, pattern matching, string matching, Deep Packet Inspection (DPI), Intrusion Detection System (IDS).

1. Introduction
Network intrusions pose a significant threat to network security. An intrusion detection system (IDS) automates the intrusion detection process by monitoring the packets coming from outside a computer system or network, and analyzes them for signs of a variety of possible attacks or probes. If these problematic packets can be detected in time, it is then possible to stop them from getting inside the network, thus preventing numerous attacks. This detection can be done by inspecting the payload of a network packet and comparing it to a collection of suspicious patterns such as Snort's. Snort is an open source network intrusion detection systems (NIDS) which can perform real-time traffic analysis [1].

The Aho-Corasick (AC) algorithm [2] is used in Snort 2.9.7 [3] and performs multiple keywords pattern matching. It can be divided into two distinct steps. The first step is to build a finite state automaton. For that, a trie or a prefix tree with input words must be constructed. The second step consists of the search of patterns itself. This search involves traveling through the automaton's nodes following transitions according to the values of input character stream. The matches are retrieved from the state of the automaton itself.

While implementing the AC algorithm, there are two key objectives: maximize the throughput, or in other words, minimize the number of cycles per character; and minimize the memory requirements of the nodes. In this paper, we propose a way of selecting a node's configuration between two types of nodes in order to achieve a trade-off between these two objectives. Our main contribution is to show that it is possible to select a parameter to obtain an optimal *memory* × *cycle* product.

The rest of the paper is organized as follows. We present the node configuration and our metric in Section 2. The methods used are discussed in Section 3. Section 4 presents the results.

2. Node configuration selection
In this section we present the two types for the node configuration that we propose. We then show how the AC nodes are distributed for Snort 2.9.7, which leads to optimal node type selection. We also present a performance metric.

2.1 Types of nodes
The two types of nodes that we propose are similar to the ones used in Shenoy's hybrid storage [4]. The difference resides in the way to store the pointers to the next nodes. The type 1 node is shown in Figure 1. It is similar to Bremler-Barr's symbol-state pairs array [5]. It consists of an array of *r* structures which is equal to the number of next nodes. Each structure is composed of two fields. The first field is the character to be matched to reach a next node. The second field is a pointer to the corresponding next node. This type of node is thus compact. A search for a match within the array of characters is linear in the worst case. It can be sped up with a binary search, at the cost of additional computation. In this work, we do not consider this type of search.

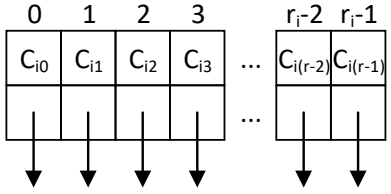

Figure 1 - Type 1 node

The type 2 node is shown in Figure 2. It is similar to Bremler-Barr's lookup tables [5]. It consists of an array of pointers, one for each

possible input character. We select an array size of 256, which corresponds to the characters of an extended ASCII table or UTF-8. The character values are used as index to a pointer to the next node if it exists. If there is no corresponding next node, the pointer value is set to NULL. This represents wasted memory. However, a search within the array of pointers can be done in constant time.

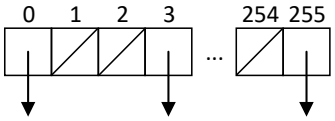

Figure 2 - Type 2 node

2.2 Node distribution and trie construction

Figure 3 shows the distribution of the number of next nodes for a trie built with the Snort 2.9.7 rule set. Almost 92% of the nodes have a single next node. For clarity of presentation, the root node with 254 next nodes is not included in the figure. For numbers of next nodes between 0 and 20, the number of nodes reduces from around a million to ten. Then, the number of nodes stabilizes in between 1 and 10. The first question for the selection of node type thus arises: at what number of next nodes should type 2 be selected instead of type 1? A compromise must be made between overall storage space and time of trie traversal.

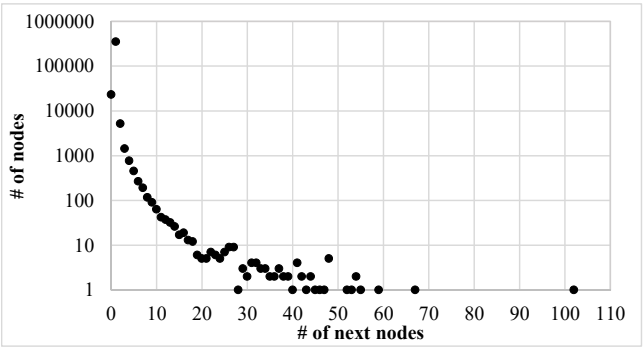

Figure 3 - Number of nodes with a specific number of next nodes

For our experiments, we built an AC trie as follows. The trie is first built with type 2 nodes. Then for each node, the array size required for node type 1 (the number of next nodes r_i) is used to select the final node type according to a defined threshold. For up to n next nodes ($r_i \leq n$), the node will be of type 1, otherwise, it will be of type 2. For example, if we set the threshold at $n = 1$, every node with zero next node will be of type 1 without any structure. The nodes with one possible next node will be of type 1 with a single structure. The nodes with 2 or more possible next nodes will be of type 2. This can lead to significant memory waste for a node. For example, if a node has only six possible next nodes, the type 2 node will have 250 NULL pointers in its array.

In this work, we compare different trie configurations using (1), where P is the performance, C is the number of cycles per character and M is the overall memory used. This relation allows to find the best compromise between C and M.

$$P = \frac{10^{10}}{C \times M} \tag{1}$$

3. Methods

To compare various trie configurations, we used a virtual machine with Xubuntu 14.04 and Valgrind was used as the profiler for the cycle count of the AC search.

The tries were constructed with 26328 patterns which are all the unique contents of the 31133 Snort 2.9.7 rules [3], which results in approximately 381k nodes. In order to create and search through a trie, an implementation from Kanani [6] was modified. For test data, many sources were used including real inputs traffics, Internet pages, the Snort rules themselves and some randomly generated character sets. All these inputs have different character counts.

4. Results

A comparison was made between three node combinations with a real traffic test set. In the first case we use only type 1 nodes, which corresponds to a value of $n = 256$. In Figure 4, this is the rightmost point and the metric has a value of 1.48. In the second case, we use only type 2 nodes, which corresponds to a value of $n = -1$. In Figure 4, this is the leftmost point in the curve and the metric has a value of 1.53. In the thirds case, we use a combination of type 1 and type 2 nodes. In Figure 4, this corresponds to all the other points on the curve and various values of n between 0 and 102. The highest value of the metric is achieved for $n = 4$ with a value of 18.51. The performance improvement in this case is approximately 12× compared with the single node-type cases. Thus, it is possible to find a balance between the two node types that trades memory space with search time.

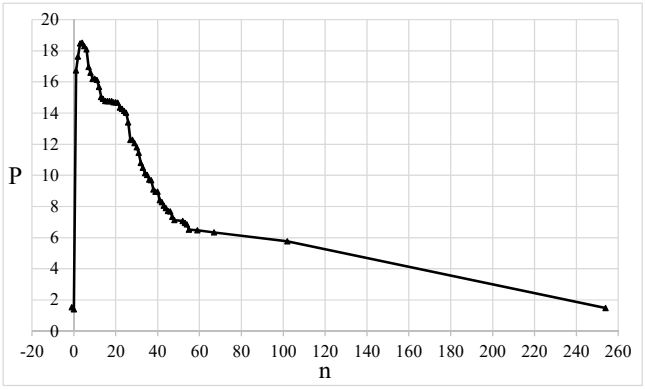

Figure 4 - Performance comparison

5. Acknowledgments

The authors would like to thank Thomas Luinaud for providing support and ideas used in this work.

6. References

[1] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX Conference on System Administration*, Berkeley, CA, USA, 1999.

[2] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM,* vol. 18, no. 6, pp. 333-340, June 1975.

[3] Snort, "Snort," 2014. [Online]. Available: https://www.snort.org/.

[4] G. Shenoy, J. Tubella and A. Gonzalez, "A Performance and Area Efficient Architecture for Intrusion Detection Systems," in *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2011.

[5] A. Bremler-Barr, Y. Harchol and D. Hay, "Space-time tradeoffs in software-based Deep Packet Inspection," in *IEEE 12th International Conference on High Performance Switching and Routing (HPSR)*, 2011.

[6] K. Kanani, "Aho-Corasick implementation (part of multifast)," 2013. [Online]. Available: http://multifast.sourceforge.net/.